# COMS W4111: Introduction to Databases
# Spring 2024, Sections 002/V02

*Homework 3*

## Introduction

- This notebook contains HW3. **Both Programming and Nonprogramming tracks should complete this homework.**
- You will submit **PDF and ZIP files** for this assignment. Gradescope will have two separate assignments for these.
- For the PDF:
  - The most reliable way to save as PDF is to go to your browser's menu bar and click `File -> Print`. Switch the orientation to landscape mode, and hit save.
  - **MAKE SURE ALL YOUR WORK (CODE AND SCREENSHOTS) IS VISIBLE ON THE PDF. YOU WILL NOT GET CREDIT IF ANYTHING IS CUT OFF.** Reach out for troubleshooting.
  - **MAKE SURE YOU DON'T SUBMIT A SINGLE PAGE PDF.** Your PDF should have multiple pages.
- For the ZIP:
  - Zip a folder containing this notebook and any screenshots.
  - You may delete any unnecessary files, such as caches.

## Setup

In [1]:
```
%load_ext sql
%sql mysql+pymysql://root:dbuserdbuser@localhost
%sql SELECT 1
```

```
 * mysql+pymysql://root:***@localhost
1 rows affected.
```

Out[1]:

| 1 |
|---|
| 1 |

In [2]:
```
%%sql

drop schema if exists s24_hw3;
create schema s24_hw3;
use s24_hw3;
```

```
 * mysql+pymysql://root:***@localhost
4 rows affected.
1 rows affected.
0 rows affected.
```

Out[2]: []

In [3]:
```
import copy
import math

import pandas
import pymysql
from sqlalchemy import create_engine

sql_conn = pymysql.connect(
    user="root",
    password="dbuserdbuser",
    host="localhost",
    port=3306,
    cursorclass=pymysql.cursors.DictCursor,
    autocommit=True
)
engine = create_engine("mysql+pymysql://root:dbuserdbuser@localhost")

cur = sql_conn.cursor()
res = cur.execute("SELECT 1")
res = cur.fetchall()
res
```

Out[3]: [{'1': 1}]

In [4]:
```
from IPython.display import Image
```

---

# Written

- As usual, try to keep things short. Do not bloviate.
- You may use external resources, but you should cite your sources.

## W1

Explain and list some differences between

- RAM
- Solid state drives
- Hard drives

*Note:* I consulted ChatGPT for discussion of the implementation of each memory type.

- **RAM** (Random Access Memory) is part of the computer's main memory and stores data available to the CPU for active processing. It is the fastest media and is **volatile**: if the system crashes or the power turns off, the contents of the RAM are lost. **RAM** is typically implemented with **volatile semi-conductor** memory techniques.
- **Solid state drives (SSD)** use flash memory (NOR Flash or NAND Flash) internally and do not use any moving parts, but provide a similar interface to a magnetic disk (such as a Hard Drive). SSD is a non-volatile form of memory preserving its contents even in the event of system crash or power failure. **SSD** is typically implemented with **non-volatile semi-conductor** memory techniques using NAND flash.
- **Hard drives (HDD)** uses magnetic disk storage and are historically the primary medium for long term storage of data. HDD is a non-volatile form of memory preserving its contents even in the event of system crash or power failure. **HDD** are implemented with **spinning magnetic disks/platters**.

**Summary (in table form):**

| | "Implementation type" | "Preservation type" |
|---|---|---|
| **RAM** | Semi-conductor | **Volatile** |
| **SSD** | Semi-conductor | **Non-Volatile** |
| **HDD** | Magnetic Disk | **Non-Volatile** |

# W2

With regards to disk drives, define

- Seek time
- Rotational latency time
- Transfer time/data transfer rate

**All of the terms** relate to components of the amount of time/**delay for I/O operations on disks** (and in particular Hard Drives).

- **Seek time** is the amount of time for the disk head to move into position over the correct cylinder/track. (Seek time is applicable only to disk drives with a moving disk head, such as a Hard Disk Drives)
- **Rotational latency time** is the amount of time for the correct sector (of the track) to rotate into position underneath the disk head. (Rotational latency time is applicable only to disk drives with a rotating disk, such as a Hard Disk Drives).
- **Transfer time/data transfer rate** refers to the amount of time to move data from the disk to memory (RAM) and vice versa. Namely, the amount of time to retrieve data from the disk or store data onto disk.

# W3

Explain the concepts of

- Logical block addressing

- Cylinder-head-sector addressing

**Note**: Parts of this answer was adapted from the lecture 8 slides.

The unit of transfer from a "disk" to the computer main memory is a "block" containing a relatively large number of bytes (e.g. 16 KB, 32 KB ...). A program requiring a byte of information must read into memory the entire block containing that byte.

- **Logical block addressing** refers to the logical location of the data in storage. The logical block addresses are typically simple and of form (DeviceID, blockID). The numbering of blockIDs is typically simple such as $1,2,3\cdots$.
- **Cylinder-head-sector addressing** is a form of physical address and refers to the physical location of the block on the disk of a **HDD**. On an **HDD** blocks are stored on magnetic disks stacked on top of each other, with multiple stacked disk heads used to read/write to the disk. The physical address for a block on an HDD therefore refers to a distinct cylinder, head and sector:
  - **Cylinder** describes the **radial distance** from the center: the disks are divided into **cylinders** or tracks radially outwards from the center of the disk.
  - The **head** describes the **"vertical distance"** by specifying **which of the stacked magnetic disks** to use: the heads are vertically stacked and therefore specifying which head acts to identify a specific disk.
  - The **sector** describes **"angular distance"** from the center of the disk, specifying which **portion of the cylinder** to use. Taken together the cylinder, head and sector therefore identify the physical address of block on a HDD.

The disk controller and disk implementation translate the **logical block address** into the **physical address** (including **Cylinder-head-sector addressing**). While the **physical address** may change over time, the block can still be retrieved by programs as the **logical block address** stays the same.

# W4

Define and list some benefits of

- Fixed-length records
- Variable-length records
- Row-oriented storage
- Column-oriented storage

- **Fixed-length records** is when each data record (tuple/row of the table) is alloted a fixed amount of storage space on disk (even when the actual record may only need less space). A benefit of this file organization is that it is **easy to access desired records** (e.g. if each record is alloted $n$ bytes, then record $i$ starts at byte $n \cdot (i-1)$.
- **Variable-length records** is when each data record (tuple/row of the table) is alloted a variable amount of storage space on disk corresponding to the actual disk space required for that particular record. A benefit of this file organization is savings on disk usage when the record contains **fields of variable length** such as strings/VARCHAR (as only the necessary space is used, as opposed to the maximum for each variable length field)
- **Row-oriented storage** is when the table is stored on disk row wise. Namely, each data record (tuple/row of a table) is stored on disk separately. This setup is beneficial (more efficient in terms of reduced I/O operations) when common queries are operations on individual rows. Example beneficial cases are when the most common queries are **transaction operations** which require performing update/insert/delete to an entire row as one unit.
- **Column-oriented storage** is when the table is stored on disk column wise. Namely, each attribute ("column") of a table is stored on disk separately. This setup is beneficial (more efficient in terms of reduced I/O operations) when common queries are operations on individual columns as only the required columns are transferred into system memory. Example beneficial cases are when the most common queries are **aggregation operations** over individual columns/projection over a subset of columns). Additional benefits are cases of vector processing.

# W5

Explain and list some differences between

- RAID 0
- RAID 1
- RAID 5

**Note**: I refered to the textbook 12.5 and the lecture slides in perparing this question.

- **RAID 0** uses multiple disks as a **single "fast" disk**. For example, if two disks are being used in RAID 0, then the data is split (striped) into two portions with one stored in each disk. In this example, the RAID 0 combined disk can perform twice as many operations at once (such as seek, write, read). If the implementation places half of the blocks of the data in each disk, then any operation requiring more than one block will be twice as fast.

- **RAID 1** uses **multiple disks as a single "reliable" disk**. Each disk contains an independent (and redundent) copy of the same data. If one disk fails, the combined RAID 1 disk is "reliable" as it still has a copy of the data on the other disks.

- **RAID 5** uses **multiple disks as a single "fast" and "reliable" disk**. RAID 5 operates by splitting the data into blocks across multiple disks and also storing an error correction block that can be used to recover lost data (e.g. a bitwise parity check block). For example, if 5 disks are being used in RAID 5, then each piece of the data is split into 4 blocks stored on separate disks and the 5th disk stores the parity block. The combined RAID 5 disk is **faster** as 4 times as many I/O operations can occur at once. The RAID 5 disk is **reliable** as if any disk fails the parity check can be used with the remaining disks to recover the value stored on the failed disk. In the event that the parity block fails, the original data is still intact on the other disks.

**Summary (in table form):**

| | "Fast" | "Reliable/Redundancy" |
|---|---|---|
| **RAID 0** | ✓ | ✗ |
| **RAID 1** | ✗ | ✓ |
| **RAID 5** | ✓ | ✓ |

# SQL

## Overview

- The `data` directory contains a file `People.csv`. The columns are
  - `nameFirst`
  - `nameLast`
  - `birthYear`
  - `birthCountry`
  - `deathYear`
  - `deathCountry`
- For Nonprogramming students, note that this `People.csv` differs from the one you loaded in HW2. Do not mix the two files.
- **There is no one right answer for this section.** You can come up with and document your own design (as long as they satisfy the requirements).

## Create Table

- Create a table based on the structure of `People.csv`
  - **You must add an additional attribute, `personID`, which has type char(9)**
    - `personID` should be the primary key of your table
  - `nameFirst` and `nameLast` cannot be null. The other (non-PK) columns can be null.
  - You should choose reasonable data types for the attributes
    - Do not use the `year` data type for `birthYear` or `deathYear`. The range for year is too small.
  - Your table will be empty for the next few sections. We will insert data later.

In [5]:
```
%%sql
create table people
(
    personID     CHAR(9)     not null,
    nameFirst    VARCHAR(32) not null,
```

```
    nameLast     VARCHAR(32) not null,
    birthYear    CHAR(4)     null,
    birthCountry VARCHAR(64) null,
    deathYear    CHAR(4)     null,
    deathCountry VARCHAR(64) null,
    constraint people_pk
        primary key (personID)
);
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
```

Out[5]: []

# Person ID Function

- `personID` is formed using the following rules:

1. The ID consists of three sections: `[lastSubstr][firstSubstr][number]`
2. `lastSubstr` is formed by lowercasing `nameLast`, then taking the first 5 letters. If `nameLast` is less than 5 letters, use the entire `nameLast`.
3. `firstSubstr` is formed by lowercasing `nameFirst`, then taking the first 2 letters. If `nameFirst` is less than 2 letters, use the entire `nameFirst`.
4. For a specific combination of `[lastSubstr][firstSubstr]`, `number` starts from 1 and increments. `number` should be padded to have length 2.
5. `nameFirst` and `nameLast` may contain periods ".", hyphens "-", and spaces " ". You should remove these characters from `nameFirst` and `nameLast` **before** doing the above substring processing.

- As an example, starting from an empty table, below is what `personID` would be assigned to the following names (assuming they were inserted in the order that they are shown)

| nameFirst | nameLast | personID |
|---|---|---|
| Donald | Ferguson | fergudo01 |
| David | Aardsma | aardsda01 |
| Doe | Fergue | fergudo02 |

| nameFirst | nameLast | personID |
|---|---|---|
| J. J. | Park | parkjj01 |

- Write a SQL function that generates a person ID using the above rules
  - You should determine what parameters and return type are needed
  - This function will be called by triggers in the next section. **It is up to you which logic you put in the function and which logic you put in the triggers.**
    - That is, if you plan to place the bulk of your logic in your triggers, then your function could be a few lines.
  - You may define helper functions
  - You may add additional attributes to your table if it helps

**Note:** Parts of this function were adapted from the lecture 6 example notebook.

In [6]:
```
%%sql
create
    definer = root@localhost function s24_hw3.personID(nameLast varchar(32), nameFirst varchar(32))
                                                    returns CHAR(9)
    deterministic
begin
    declare first_clean varchar(32);
    declare last_clean varchar(32);
    declare first_pattern char(2);
    declare last_pattern char(5);
    declare ID_count INT;
    declare result CHAR(9);
    declare ID_prefix CHAR(7);
    declare ID_pattern varchar(16);
    declare ID_suffix CHAR(3);

    /* remove punctuation and spaces from names */
    set first_clean = REPLACE(REGEXP_REPLACE(nameFirst, '[[:punct:]]', ''),' ', '');
    set last_clean = REPLACE(REGEXP_REPLACE(nameLast, '[[:punct:]]', ''), ' ', '');


    /* Form patterns of lower case characters of first 5 letters of the last_name,
    first 2 letters of first_name */

    set last_pattern = lower(substr(last_clean, 1, 5));
```

```
    set first_pattern = lower(substr(first_clean, 1, 2));

    /* Create the ID prefix and pattern.   */
    set ID_prefix = concat(last_pattern, first_pattern);
    set ID_pattern = concat(ID_prefix, '[0-9]*');

    /* Initialize ID_count */
    set ID_count = 0;

    /* Find the maximum ID value for the pattern
       Keep only the numerical suffix */
    select substr(IFNULL(MAX(personID),'100'),-2) into ID_count
    from people
    where REGEXP_LIKE(personID,ID_pattern);

    /* Concat with prefix */
    IF ID_count < 9 then
        set result = concat(ID_prefix,0, ID_count+1);
    Else
        set result = concat(ID_prefix, ID_count+1);
    END IF;

    /* Return the result */
    return result;
end;
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
```

Out[6]: []

## Insert and Update Triggers

- We want to automatically generate `personID` using the function above whenever a row is inserted. The user should not need to manually specify it.
- Write a SQL trigger that runs every time a row is inserted
  - The trigger should generate a person ID for the row based on its `nameFirst` and `nameLast`; it should then set the `personID` for that row.

    - This should occur even if the user attempts to manually set `personID`. The user's value for `personID` is ignored.
    - You should call the function you wrote above
- Write another SQL trigger that runs every time a row is updated
    - There is no `immutable` keyword in MySQL; however, we can simulate immutability using a trigger. If the user attempts to modify `personID` directly, throw an exception.
    - If the user modifies `nameFirst` or `nameLast` such that the `personID` is no longer valid based on the rules in the previous section (specifically, if `[lastSubstr][firstSubstr]` is no longer the same as before), you should re-generate `personID` and re-set it.
        - You should call the function you wrote above
- **You are writing two SQL triggers for this section**

In [7]:
```
%%sql
create definer = root@localhost trigger s24_hw3.set_person_id
    before insert
    on s24_hw3.people
    for each row
begin
    set new.personID = personID(new.nameLast, new.nameFirst);
    set @insertID = new.personID;
end;

create definer = root@localhost trigger s24_hw3.update_person_id
    before update
    on s24_hw3.people
    for each row
begin
    Declare potential Char(9);
    Select personID(IFNULL(new.nameLast,old.nameLast), IFNULL(new.nameFirst,old.nameFirst))
            into potential;
    if substr(old.personID, 1, Length(old.personID)-2)
        != substr(potential, 1,  Length(old.personID)-2) then
        set new.personID = potential;
        set @updateID = new.personID;

    elseif old.personID != IFNULL(new.personID, old.personID) then
        signal sqlstate '02000'
```

```
            set message_text = "You cannot change the personID";
        set new.personID = old.personID;
        set @updateID = new.personID;

    else
        set new.personID = old.personID;
        set @updateID = new.personID;
    end if;
end;
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
0 rows affected.
```

Out[7]: []

## Create and Update Procedures

- You must implement two stored procedures

1. `createPerson(nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry, personID)`
   A. `personID` is an out parameter. It should be set to the ID generated for the person.
   B. All the other parameters are in paramaters
2. `updatePerson(personID, nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry, newPersonID)`
   A. `newPersonID` is an out parameter. It should be set to the ID of the person after the update (even if it didn't change).
   B. All the other parameters are in parameters.
      a. `personID` is used to identify the row that the user wants to update. The other in parameters are the values that the user wants to set.
      b. **Ignore null in parameters.** Only update an attribute if the in parameter is non-null.

- Depending on how you implemented your triggers, these procedures could be as simple as calling `insert` / `update` and setting the out parameters

In [8]:
```
%%sql
create definer = root@localhost
    PROCEDURE
    createPerson(
    IN name_first VARCHAR(32), name_Last VARCHAR(32), birth_Year CHAR(4), birth_Country VARCHAR(64),
                death_Year CHAR(4), death_Country VARCHAR(64), OUT person_ID CHAR(9))
    BEGIN
        INSERT
        into s24_hw3.people(personID,nameFirst,
                            nameLast,birthYear,birthCountry,deathYear,deathCountry)
            values(person_ID,name_first,
                   name_Last,birth_Year,birth_Country,death_Year,death_Country);

        SET person_ID = @insertID;
    end;

create definer = root@localhost
    PROCEDURE
    updatePerson(
    IN person_ID CHAR(9), name_First VARCHAR(32), name_Last VARCHAR(32),
       birth_Year CHAR(4), birth_Country VARCHAR(64),
       death_Year CHAR(4), death_Country VARCHAR(64), OUT newPersonID CHAR(9))

    BEGIN
        UPDATE s24_hw3.people
        SET people.nameFirst = IFNULL(name_First, people.nameFirst),
            people.nameLast = IFNULL(name_Last,people.nameLast),
            people.birthYear = IFNULL(birth_Year, people.birthYear),
            people.birthCountry = IFNULL(birth_Country,people.birthCountry),
            people.deathYear = IFNULL(death_Year, people.deathYear),
            people.deathCountry = IFNULL(death_Country,people.deathCountry),
            people.personID = personID
        WHERE people.personID=person_ID;
        SET newPersonID = @updateID;
    end;
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
0 rows affected.
```

Out[8]: []

## Security

- You must create a new user `general_user` and use security to allow it to perform only `select` and `execute` operations (i.e., no `insert`, `delete`, and `update` operations)

In [9]:
```
%%sql
DROP USER IF EXISTS 'general_user'@'%';
CREATE USER 'general_user'@'%' IDENTIFIED BY 'dbuserdbuser';
GRANT SELECT, EXECUTE on s24_hw3.* to 'general_user'@'%';
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
0 rows affected.
0 rows affected.
```

Out[9]: []

## Inheritance Using Views

- A person can be a player or manager
  - That is, a player is-a person, and a manager is-a person
- Describe how you could implement this inheritance relationship given that you already have your `people` table
  - No code is necessary

As we already have the `people` table, we can implement the above inheritance relationship using **views** and the **"3 table"** specialization method.

We use the following 3 tables:

1. The `people` table that we already have contains the common attributes for all people (including players and managers).
2. A new table `player_table` is created. The `player_table` table contains the `personID` attribute as its Primary key and as a Foreign key referencing the PK of the `people` table. The `player_table` table additionally contains all attributes that are unique to players.

3. Similar to (2), a new table `manager_table` is created. The `manager_table` table contains the `personID` attribute as its Primary key and as a Foreign key referencing the PK of the `people` table. The `manager_table` table additionally contains all attributes that are unique to managers.

Finally, we can create the **view** `player` from the **join** of `people` $\bowtie_{personID}$ `player_table` ; and the **view** `manager` from the **join** of `people` $\bowtie_{personID}$ `manager_table` . Namely:

- The `player` view contains all common attributes ("inherited" from `people` ) and the attributes unique to players (from `player_table` ).
- Similarly, the `manager` view contains all common attributes ("inherited" from `people` ) and the attributes unique to managers (from `manager_table` ).

## Data Insertion Testing

- The cells below load data from `People.csv` to your database
  - No code is required on your part. Make sure everything runs without error.

In [10]:
```
# Load People.csv into a dataframe.
# You may see NaNs in the non-null columns. This is fine.

people_df = pandas.read_csv("data/People.csv")
people_df.head(10)
```

Out[10]:

| | nameFirst | nameLast | birthYear | birthCountry | deathYear | deathCountry |
|---|---|---|---|---|---|---|
| **0** | Ed | White | 1926.0 | USA | 1982.0 | USA |
| **1** | Sparky | Adams | 1894.0 | USA | 1989.0 | USA |
| **2** | Bob | Johnson | 1959.0 | USA | NaN | NaN |
| **3** | Johnny | Ryan | 1853.0 | USA | 1902.0 | USA |
| **4** | Jose | Alvarez | 1956.0 | USA | NaN | NaN |
| **5** | Andrew | Brown | 1981.0 | USA | NaN | NaN |
| **6** | Chris | Johnson | 1984.0 | USA | NaN | NaN |
| **7** | Johnny | Johnson | 1914.0 | USA | 1991.0 | USA |
| **8** | Albert | Williams | 1954.0 | Nicaragua | NaN | NaN |
| **9** | Ed | Brown | NaN | USA | NaN | NaN |

In [11]:

```
def add_person(p):
    """
    p is a dictionary containing the column values for either a student or an employee.
    """

    cur = sql_conn.cursor()

    # This function changes the data, converting nan to None.
    # So, we make a copy and change the copy.
    p_dict = copy.copy(p)
    for k, v in p_dict.items():
        if isinstance(v, float) and math.isnan(v):
            p_dict[k] = None

    # This provides a hint for what your stored procedure will look like.
    res = cur.callproc("s24_hw3.createPerson",
                       # The following are in parameters
                        (p_dict['nameFirst'],
                        p_dict['nameLast'],
                        p_dict['birthYear'],
                        p_dict['birthCountry'],
```

```
                        p_dict['deathYear'],
                        p_dict['deathCountry'],
                        # The following are out parameters for personID.
                        None))

    # After the procedure executes, the following query will select the out values.
    res = cur.execute("""SELECT @_s24_hw3.createPerson_6""")
    result = cur.fetchall()

    sql_conn.commit()
    cur.close()
    return result[0]["@_s24_hw3.createPerson_6"]  # Return personID
```

- Below is the main data insertion logic
  - `add_person` calls your `createPerson` procedure
  - The `data` directory also contains a file `People_Ids.csv`, which is the expected `personID` for each row after it is inserted. We'll use this to check your `createPerson` implementation.

In [12]:
```
%sql truncate table s24_hw3.people

expected_ids_df = pandas.read_csv("data/People-Ids.csv", header=None)
expected_ids = [e[0] for e in expected_ids_df.values.tolist()]

for i, (p, e_id) in enumerate(zip(people_df.to_dict(orient="records"), expected_ids)):
    p_id = add_person(p)
    assert p_id == e_id, \
    f"Row {i}: Expected {e_id}, but got {p_id} for {p['nameFirst']} {p['nameLast']}"

print("Successfully inserted all data")
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
Successfully inserted all data
```

## Data Updating Testing

- The following cells test your update trigger and `updatePerson` implementation

- No code is required on your part. Make sure everything runs as expected.
- The tests assume you just finished the Data Insertion Testing section. You may run into issues if you run the Data Updating Testing section multiple times without reseting your data.

In [13]:
```
# Switch back to root
%sql mysql+pymysql://root:dbuserdbuser@localhost/s24_hw3

def transform(d):
    # %sql returns dict of attributes to one-tuples.
    # This function extracts the values from the one-tuples.
    return {k: v[0] for k, v in d.items()}

def is_subset(d1, d2):
    # Checks if d1 is a subset of a d2
    for k, v in d1.items():
        if k not in d2 or str(d2[k]) != str(v):
            return False
    return True
```

In [14]:
```
# Create new person to test on

%sql call createPerson("Babe", "Ruth", null, null, null, null, @ruthID)
res1 = %sql select * from people p where p.personID = @ruthID
res1_d = transform(res1.dict())
expected_d = dict(
    personID="ruthba01",
    nameFirst="Babe",
    nameLast="Ruth",
    birthYear=None,
    birthCountry=None,
    deathYear=None,
    deathCountry=None
)

print(res1)

assert is_subset(expected_d, res1_d), \
f"Row has unexpected value. Expected {expected_d}, but got {res1_d}"

print("Success")
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
+----------+-----------+----------+-----------+--------------+-----------+--------------+
| personID | nameFirst | nameLast | birthYear | birthCountry | deathYear | deathCountry |
+----------+-----------+----------+-----------+--------------+-----------+--------------+
| ruthba01 |    Babe   |   Ruth   |    None   |     None     |    None   |     None     |
+----------+-----------+----------+-----------+--------------+-----------+--------------+
Success
```

In [15]:

```
# Update birth country and year
%sql call updatePerson(@ruthID, null, null, 1895, "USA", 1948, "USA", @ruthID)
res2 = %sql select * from people p where p.personID = @ruthID

res2_d = transform(res2.dict())
expected_d = dict(
    personID="ruthba01",
    nameFirst="Babe",
    nameLast="Ruth",
    birthYear=1895,
    birthCountry="USA",
    deathYear=1948,
    deathCountry="USA"
)

print(res2)

assert is_subset(expected_d, res2_d), \
f"Row has unexpected value. Expected {expected_d}, but got {res2_d}"

print("Success")
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
+----------+-----------+----------+-----------+--------------+-----------+--------------+
| personID | nameFirst | nameLast | birthYear | birthCountry | deathYear | deathCountry |
+----------+-----------+----------+-----------+--------------+-----------+--------------+
| ruthba01 |    Babe   |   Ruth   |    1895   |     USA      |    1948   |     USA      |
+----------+-----------+----------+-----------+--------------+-----------+--------------+
Success
```

In [16]:
```
# Checking that null is a noop
%sql call updatePerson(@ruthID, null, null, null, null, null, null, @ruthID)
res3 = %sql select * from people p where p.personID = @ruthID
res3_d = transform(res3.dict())

print(res3)

assert is_subset(expected_d, res3_d), \
f"Row has unexpected value. Expected {expected_d}, but got {res3_d}"

print("Success")
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
+----------+-----------+----------+-----------+--------------+-----------+--------------+
| personID | nameFirst | nameLast | birthYear | birthCountry | deathYear | deathCountry |
+----------+-----------+----------+-----------+--------------+-----------+--------------+
| ruthba01 |    Babe   |   Ruth   |    1895   |     USA      |    1948   |     USA      |
+----------+-----------+----------+-----------+--------------+-----------+--------------+
Success
```

In [17]:
```
# Try to manually set personID
# Note: You should get an OperationalError. If you get an AssertionError, then
# your trigger is not doing its job.
```

```
res4 = %sql update people set personID = "dff9" where personID = "ruthba01"

assert res4 is None, "Your trigger should throw an exception"

print("Success")
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
(pymysql.err.OperationalError) (1643, 'You cannot change the personID')
[SQL: update people set personID = "dff9" where personID = "ruthba01"]
(Background on this error at: https://sqlalche.me/e/20/e3q8)
Success
```

In [18]:
```
# Check that update trigger updates personID if name changes

%sql call updatePerson(@ruthID, "George", "Herman", 1920, "USA", 2005, "USA", @ruthID)
res5 = %sql select * from people p where p.personID = @ruthID
res5_d = transform(res5.dict())
expected_d = dict(
    personID="hermage01",
    nameFirst="George",
    nameLast="Herman",
    birthYear=1920,
    birthCountry="USA",
    deathYear=2005,
    deathCountry="USA"
)

print(res5)

assert is_subset(expected_d, res5_d), \
f"Row has unexpected value. Expected {expected_d}, but got {res5_d}"

print("Success")
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
+-----------+-----------+----------+-----------+--------------+-----------+--------------+
|  personID | nameFirst | nameLast | birthYear | birthCountry | deathYear | deathCountry |
+-----------+-----------+----------+-----------+--------------+-----------+--------------+
| hermage01 |   George  |  Herman  |    1920   |     USA      |    2005   |     USA      |
+-----------+-----------+----------+-----------+--------------+-----------+--------------+
Success
```

## Security Testing

- Write and execute statements below to show that you set up the permissions for `general_user` correctly
  - You should show that `select` and `execute` work, but `insert`, `update`, and `delete` don't

In [19]:
```
# Connect to database as general_user
%sql mysql+pymysql://general_user:dbuserdbuser@localhost/s24_hw3
```

In [20]:
```
#test to show that select works for general_user
res6 = %sql select * from people p where p.personID = "hermage01"
res6_d = transform(res6.dict())
expected_d = dict(
    personID="hermage01",
    nameFirst="George",
    nameLast="Herman",
    birthYear=1920,
    birthCountry="USA",
    deathYear=2005,
    deathCountry="USA"
)

print(res6)

assert is_subset(expected_d, res6_d), \
f"Row has unexpected value. Expected {expected_d}, but got {res6_d}"

print("Success")
```

```
 * mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
   mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
+-----------+-----------+----------+-----------+--------------+-----------+--------------+
|  personID | nameFirst | nameLast | birthYear | birthCountry | deathYear | deathCountry |
+-----------+-----------+----------+-----------+--------------+-----------+--------------+
| hermage01 |   George  |  Herman  |    1920   |     USA      |    2005   |     USA      |
+-----------+-----------+----------+-----------+--------------+-----------+--------------+
Success
```

In [21]:
```
#test to show that execute works for general_user, e.g. for create person
%sql CALL createPerson("Darth", "Vader", null, null, null, null, @vaderID);
res7 = %sql select * from people p where p.personID = @vaderID
res7_d = transform(res7.dict())
expected_d = dict(
    personID="vaderda01",
    nameFirst="Darth",
    nameLast="Vader",
    birthYear=None,
    birthCountry=None,
    deathYear=None,
```

```
    deathCountry=None
)

print(res7)

assert is_subset(expected_d, res7_d), \
f"Row has unexpected value. Expected {expected_d}, but got {res7_d}"

print("Success")
```

```
 * mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
   mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
 * mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
   mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
+-----------+-----------+----------+-----------+--------------+-----------+--------------+
|  personID | nameFirst | nameLast | birthYear | birthCountry | deathYear | deathCountry |
+-----------+-----------+----------+-----------+--------------+-----------+--------------+
| vaderda01 |   Darth   |  Vader   |    None   |     None     |    None   |     None     |
+-----------+-----------+----------+-----------+--------------+-----------+--------------+
Success
```

In [22]:

```
#test to show that execute works for general_user, e.g. for update person
%sql CALL updatePerson(@vaderID,"Darth", "Vader", 1977, "Tatooine", 1983, "Endor", @vaderID);
res8 = %sql select * from people p where p.personID = @vaderID
res8_d = transform(res8.dict())
expected_d = dict(
    personID="vaderda01",
    nameFirst="Darth",
    nameLast="Vader",
    birthYear=1977,
    birthCountry="Tatooine",
    deathYear=1983,
    deathCountry="Endor"
)

print(res8)

assert is_subset(expected_d, res8_d), \
```

```
f"Row has unexpected value. Expected {expected_d}, but got {res8_d}"

print("Success")
```

```
 * mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
   mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
 * mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
   mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
+-----------+-----------+----------+-----------+--------------+-----------+--------------+
|  personID | nameFirst | nameLast | birthYear | birthCountry | deathYear | deathCountry |
+-----------+-----------+----------+-----------+--------------+-----------+--------------+
| vaderda01 |   Darth   |  Vader   |    1977   |   Tatooine   |    1983   |    Endor     |
+-----------+-----------+----------+-----------+--------------+-----------+--------------+
Success
```

In [23]:
```
#test showing that insert does not work for general_user
# Note: As above, we want to get an OperationalError.
res9 = %sql INSERT INTO people (personID, nameFirst, nameLast, birthYear,\
                                birthCountry, deathYear, deathCountry)\
VALUES ("jedi01","Obi Wan", "Kenobi",null,null,null,null)

assert res9 is None, "There should be an OperationalError"

print("Success")
```

```
 * mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
   mysql+pymysql://root:***@localhost/s24_hw3
(pymysql.err.OperationalError) (1142, "INSERT command denied to user 'general_user'@'localhost' for table
'people'")
[SQL: INSERT INTO people (personID, nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry)
VALUES ("jedi01","Obi Wan", "Kenobi" ,null,null,null,null)]
(Background on this error at: https://sqlalche.me/e/20/e3q8)
Success
```

In [24]:
```
#test showing that update does not work for general_user
# Note: We want to get an OperationalError.

res10 = %sql UPDATE people SET nameFirst = "Anakin", nameLast = "Skywalker" \
```

```
WHERE personID = "vaderda01"

assert res10 is None, "There should be an OperationalError"

print("Success")
```

```
 * mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
   mysql+pymysql://root:***@localhost/s24_hw3
(pymysql.err.OperationalError) (1142, "UPDATE command denied to user 'general_user'@'localhost' for table
'people'")
[SQL: UPDATE people SET nameFirst = "Anakin" , nameLast = "Skywalker" WHERE personID = "vaderda01"]
(Background on this error at: https://sqlalche.me/e/20/e3q8)
Success
```

In [25]:
```
#test showing that delete does not work for general_user
# Note: We want to get an OperationalError.

res11 = %sql DELETE from people where personID = "vaderda01"

assert res11 is None, "There should be an OperationalError"

print("Success")
```

```
 * mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
   mysql+pymysql://root:***@localhost/s24_hw3
(pymysql.err.OperationalError) (1142, "DELETE command denied to user 'general_user'@'localhost' for table
'people'")
[SQL: DELETE from people where personID = "vaderda01"]
(Background on this error at: https://sqlalche.me/e/20/e3q8)
Success
```

# GoT Data Visualization

## Data Loading

- Run the cell below to create and insert data into GoT-related tables

In [26]:
```
%sql mysql+pymysql://root:dbuserdbuser@localhost/s24_hw3

for filename in [
    "episodes_basics", "episodes_characters", "episodes_scenes"
]:
    df = pandas.read_json(f"data/{filename}.json")
    df.to_sql(name=filename, schema="s24_hw3", con=engine, index=False, if_exists="replace")

print("Success")
```

```
Success
```

## Overview

- In this section, you'll be combining SQL and Dataframes to create data visualizations
  - You may find this notebook helpful
  - You may also find the Pandas docs helpful
- **For all questions, you need to show the SQL output and the visualization generated from it.** See DV0 for an example.

## DV0

- This question is an example of what is required from you
- Create a bar graph showing the amount of time each season ran for (in seconds)
- You should use the `episodes_scenes` table

- Note: `season_running_time <<` in the following cell saves the output of the SQL query into a local Python variable `season_running_time`

In [27]:
```
%%sql

season_running_time <<

with one as (
    select seasonNum, episodeNum, sceneNum, sceneEnd, time_to_sec(sceneEnd) as sceneEndSeconds,
           sceneStart,  time_to_sec(sceneStart) as sceneStartSeconds,
            time_to_sec(sceneEnd)-time_to_sec(sceneStart) as sceneLengthSeconds
    from episodes_scenes
),
two as (
    select seasonNum, episodeNum, max(sceneEnd) as episodeEnd, max(sceneEndSeconds) as episodeEndSeconds
    from one
    group by seasonNum, episodeNum
),
    three as (
        select seasonNum, cast(sum(episodeEndSeconds) as unsigned) as totalSeasonSeconds,
               sec_to_time(sum(episodeEndSeconds)) as totalRunningTime
        from two
        group by seasonNum
    )
select * from three;
```

```
  mysql+pymysql://general_user:***@localhost/s24_hw3
  mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
8 rows affected.
Returning data to local variable season_running_time
```

In [28]:
```
# You must show the SQL output

season_running_time = season_running_time.DataFrame()
season_running_time
```

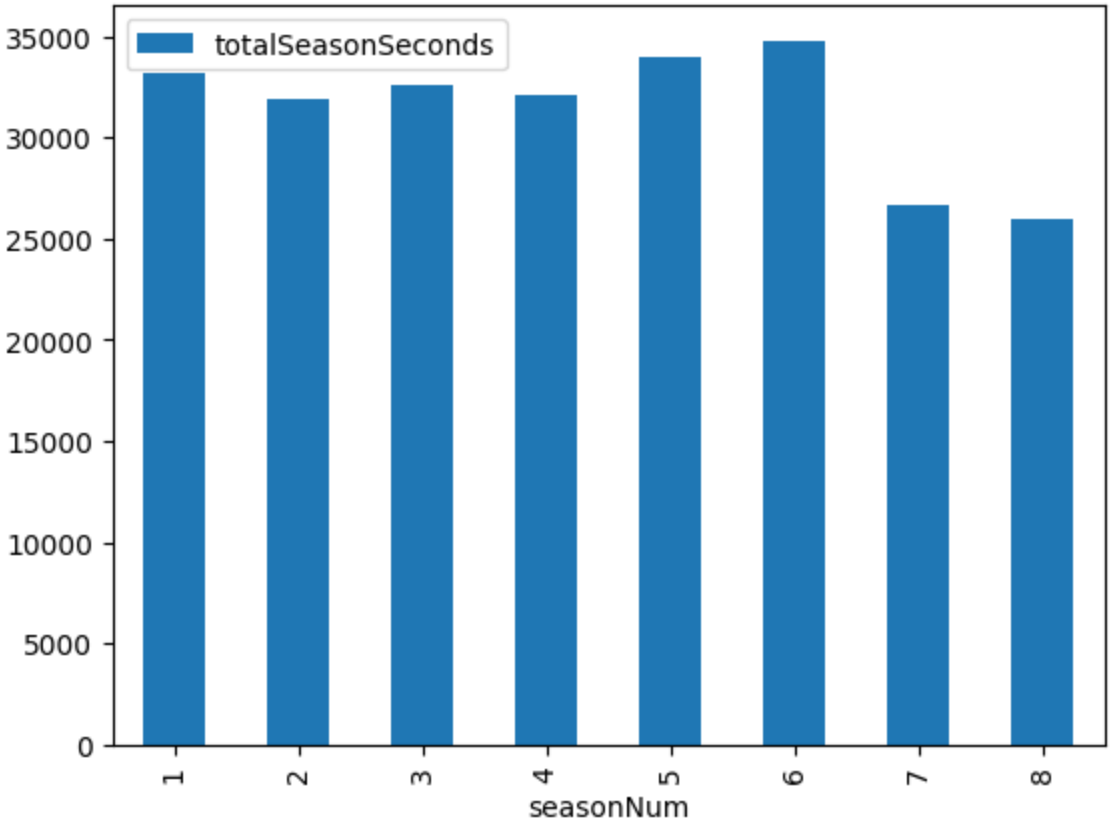
Out[28]:

| | seasonNum | totalSeasonSeconds | totalRunningTime |
|---|---|---|---|
| **0** | 1 | 33143 | 0 days 09:12:23 |
| **1** | 2 | 31863 | 0 days 08:51:03 |
| **2** | 3 | 32541 | 0 days 09:02:21 |
| **3** | 4 | 32100 | 0 days 08:55:00 |
| **4** | 5 | 34003 | 0 days 09:26:43 |
| **5** | 6 | 34775 | 0 days 09:39:35 |
| **6** | 7 | 26675 | 0 days 07:24:35 |
| **7** | 8 | 25922 | 0 days 07:12:02 |

In [29]:
```
# You must show the visualization

season_running_time[['seasonNum', 'totalSeasonSeconds']].plot.bar(x='seasonNum', y='totalSeasonSeconds')
```

Out[29]:
```
<Axes: xlabel='seasonNum'>
```

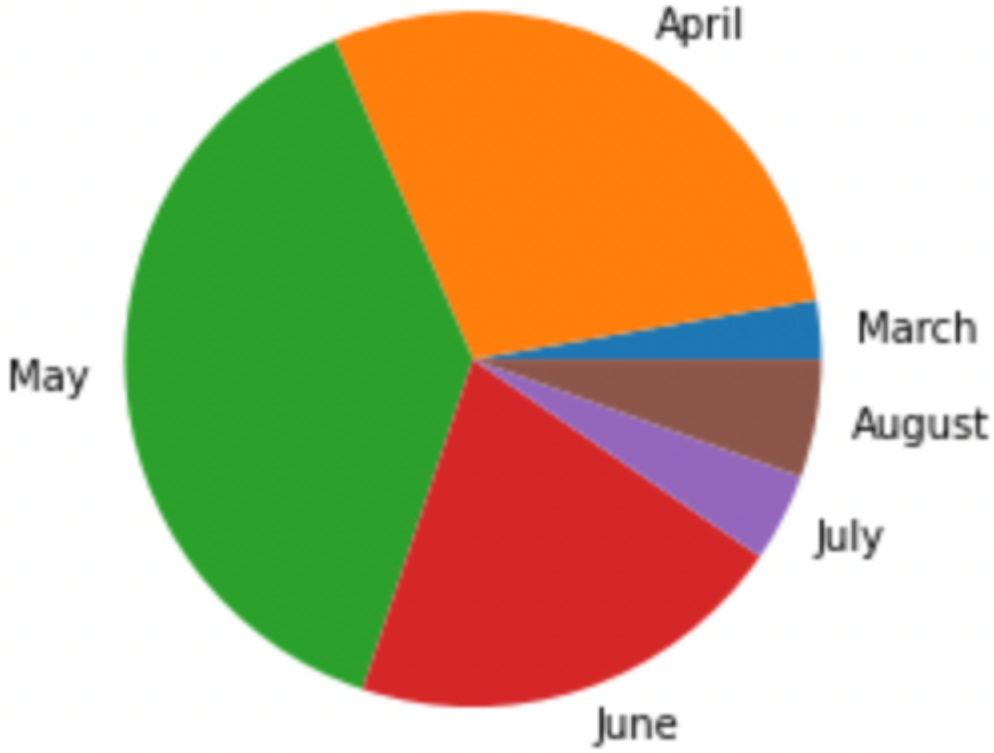



## DV1

- Create a pie chart showing the proportion of episodes aired in each month (regardless of year)
- You should use the `episodes_basics` table
- As an example, your pie chart may look like this:

In [30]:
```
Image("./images/NP1.png")
```

Out[30]:




In [31]:
```
%%sql

episodes_per_month <<
With ordered as (select MONTHNAME(episodeAirDate) as months,
                MONTH(episodeAirDate) as month_number,
                Count(*) as total from episodes_basics
                      GROUP BY months,month_number
                      ORDER BY month_number),
     output as (select months,total from ordered)
select * from output;
```

```
   mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
6 rows affected.
Returning data to local variable episodes_per_month
```

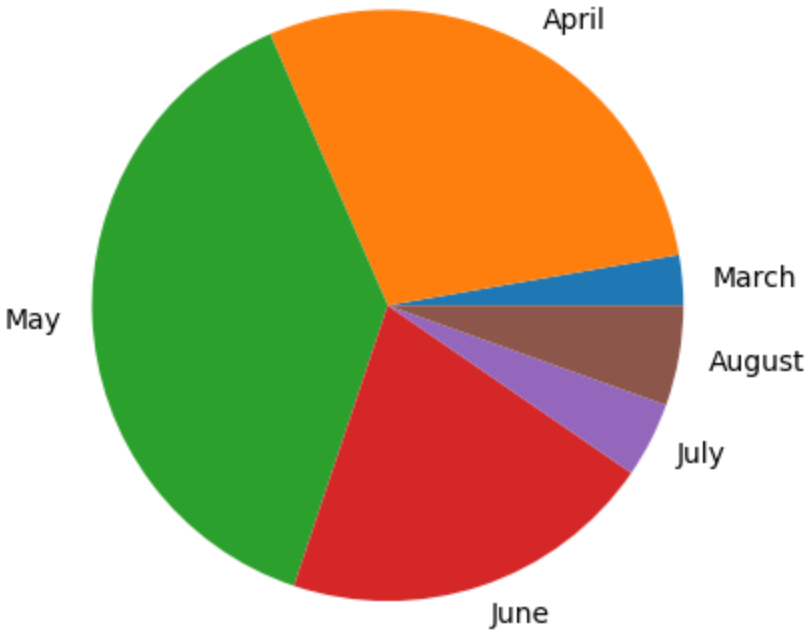
In [32]:
```
# SQL output
```

```
episodes_per_month = episodes_per_month.DataFrame()
episodes_per_month
```

Out[32]:

| | months | total |
|---|---|---|
| **0** | March | 2 |
| **1** | April | 21 |
| **2** | May | 28 |
| **3** | June | 15 |
| **4** | July | 3 |
| **5** | August | 4 |

In [33]:
```
# todo: visualization
episodes_per_month.plot.pie(labels=episodes_per_month['months'],y='total',ylabel='',legend=False);
```

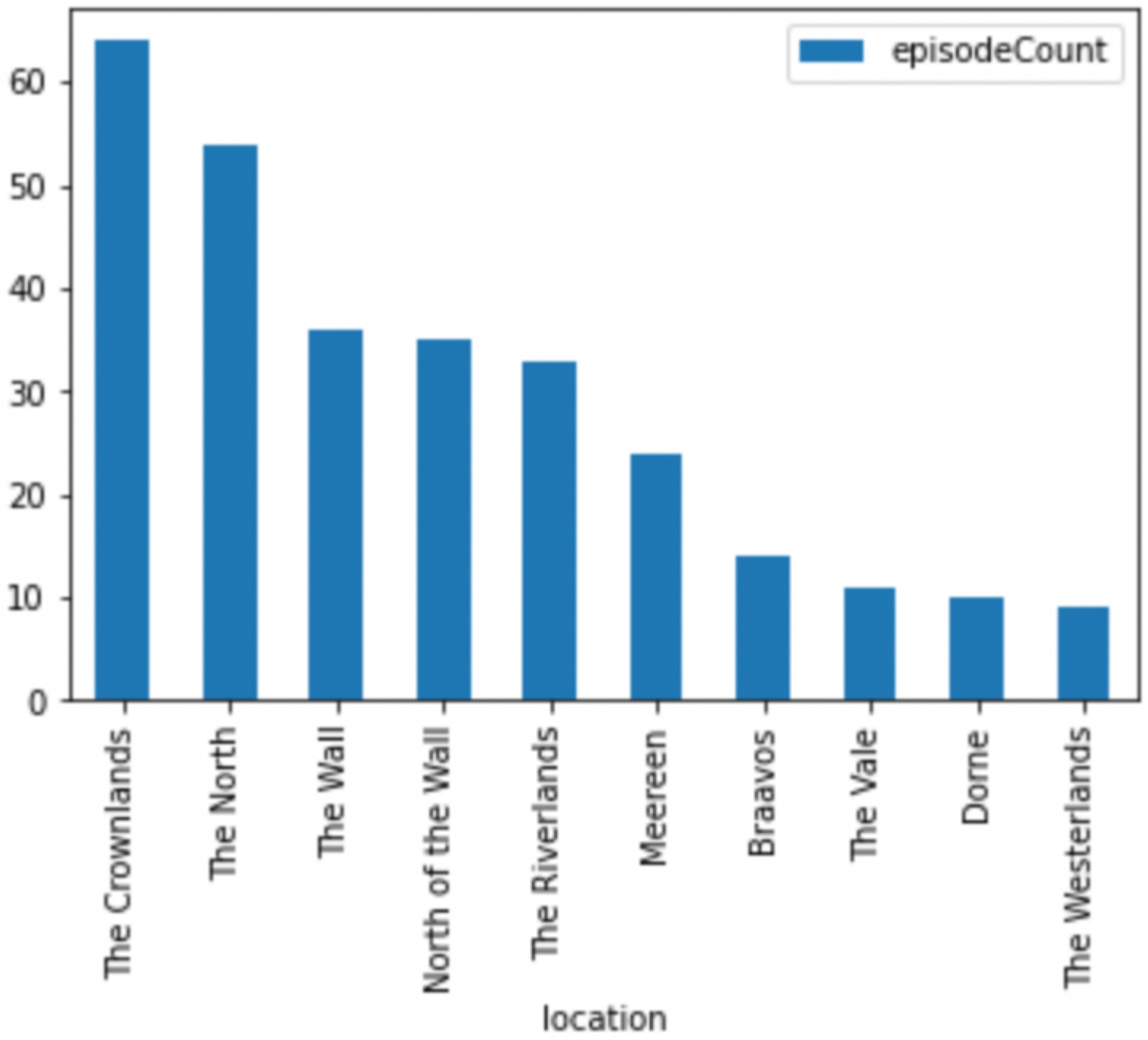



## DV2

- Create a bar chart showing the number of episodes that every location (not sublocation) appeared in
  - You are counting the number of episodes, not scenes. If a location appeared in multiple scenes in a single episode, that should increment your count only by one.
  - You should order your chart on the number of episodes descending, and you should only show the top 10 locations
- You should use the `episodes_scenes` table
- As an example, your bar chart may look like this:

In [34]:
```
Image("./images/NP3.png")
```

Out[34]:




In [35]:
```
%%sql
```

```
location_episode_count <<

select sceneLocation as location,
Count(Distinct Concat(seasonNum,episodeNum)) as episodeCount
from episodes_scenes
GROUP BY location
ORDER BY episodeCount DESC
LIMIT 10;
```

```
   mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
10 rows affected.
Returning data to local variable location_episode_count
```

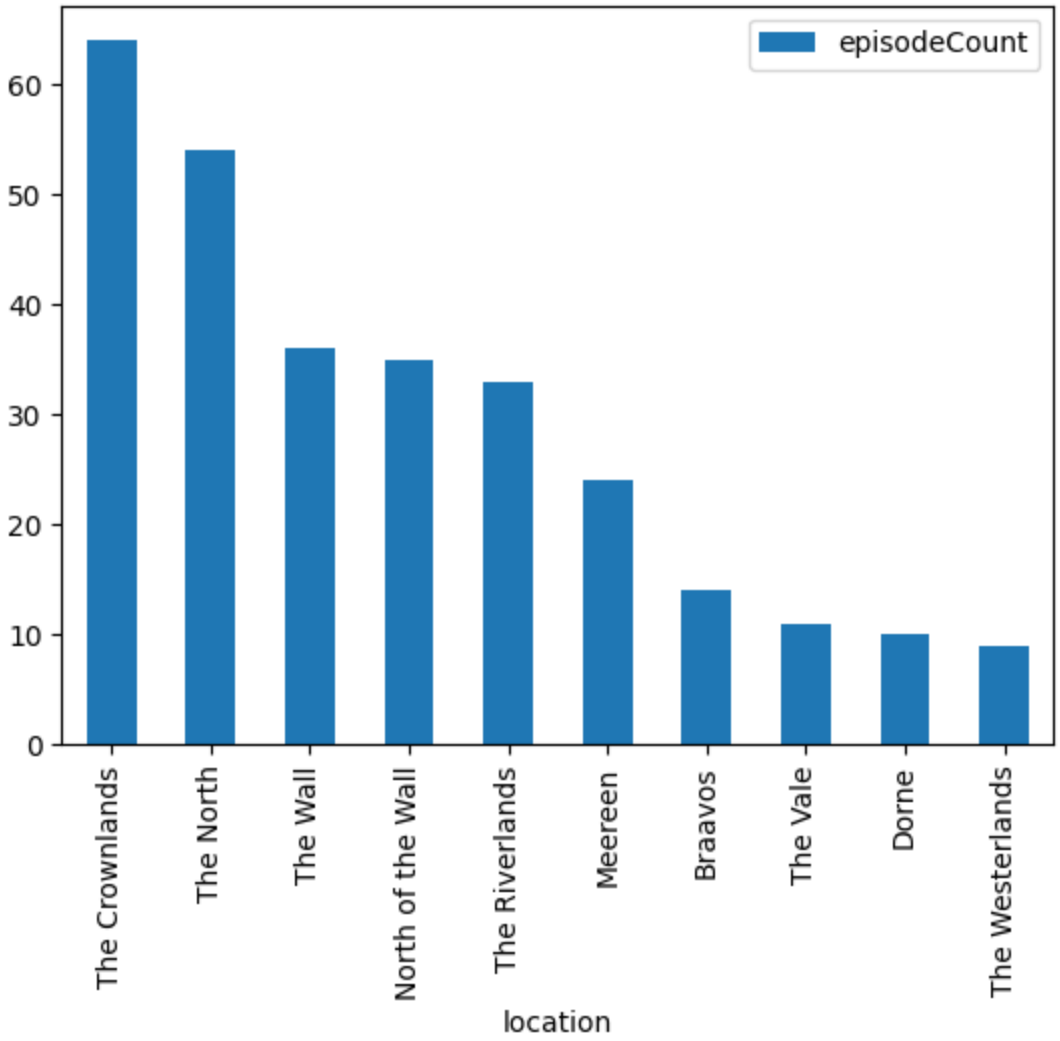
In [36]:
```
# SQL output

location_episode_count = location_episode_count.DataFrame()
location_episode_count
```

Out[36]:

| | location | episodeCount |
|---|---|---|
| **0** | The Crownlands | 64 |
| **1** | The North | 54 |
| **2** | The Wall | 36 |
| **3** | North of the Wall | 35 |
| **4** | The Riverlands | 33 |
| **5** | Meereen | 24 |
| **6** | Braavos | 14 |
| **7** | The Vale | 11 |
| **8** | Dorne | 10 |
| **9** | The Westerlands | 9 |

In [37]:
```
# todo: visualization
location_episode_count.plot.bar(x='location');
```

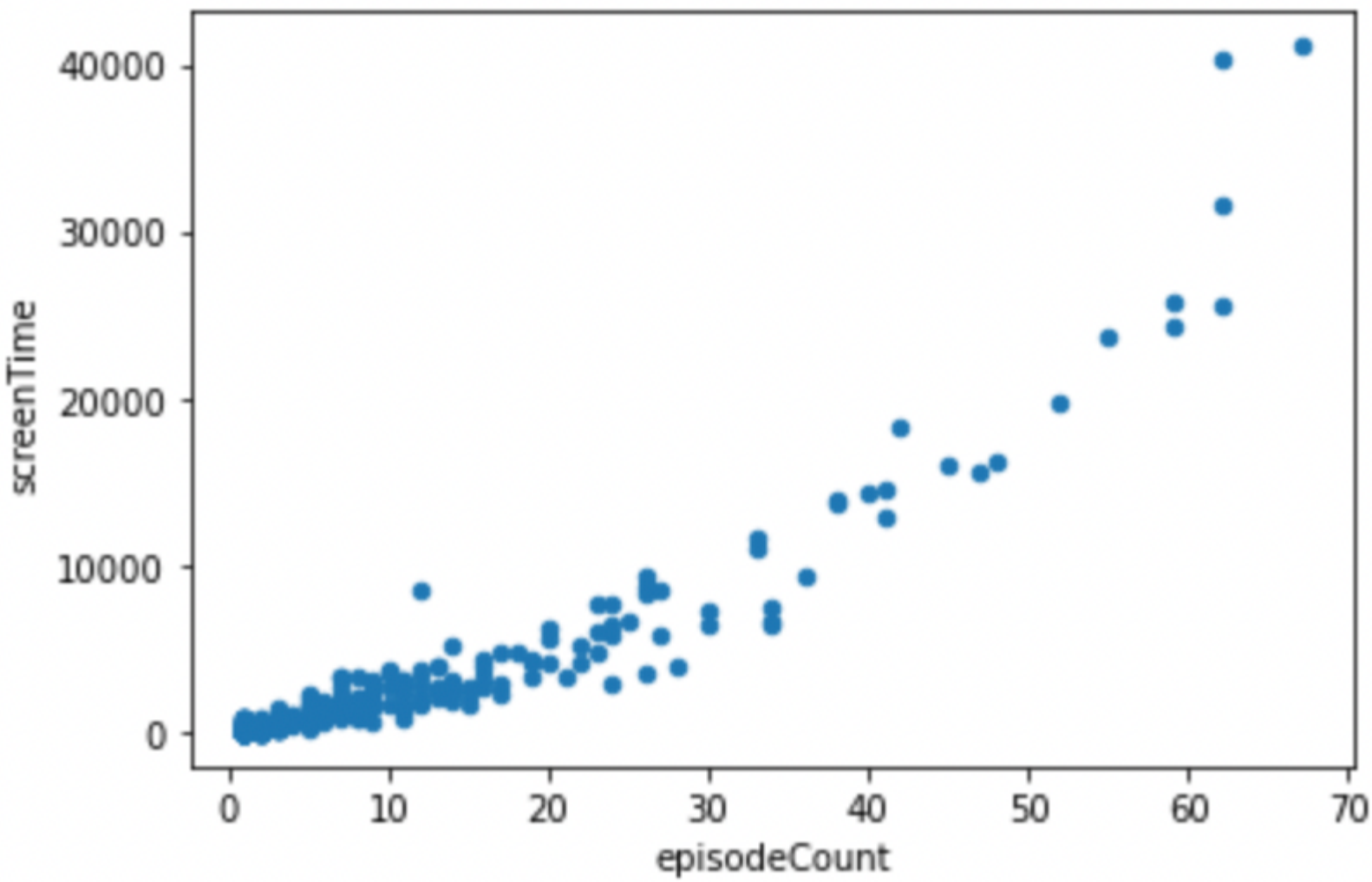

episodeCount
60
50
40
30
20
10
0
The Crownlands
The North
The Wall
North of the Wall
The Riverlands
Meereen
Braavos
The Vale
Dorne
The Westerlands
location

# DV3

- Create a scatter plot showing the relationship between the number of episodes (not scenes) a character appears in and their screen time (in seconds)
  - A character's screen time is the sum of the time lengths of all the scenes that the character appears in
- You should use the `episodes_characters` and `episodes_scenes` tables
- As an example, your scatter plot may look like this:

In [38]:
```
Image("./images/NP4.png")
```

Out[38]:




In [39]:
```
%%sql

episode_count_screen_time <<

With scene_times as (select seasonNum,
                     episodeNum,
                     sceneNum,
                     time_to_sec(sceneEnd)-time_to_sec(sceneStart) as sceneLength
    from episodes_scenes),
    character_scene_times as (select seasonNum,
```

```
                                episodeNum,
                                sceneNum,
                                sceneLength,
                                characterName
                    from scene_times join episodes_characters using(seasonNum,sceneNum, episodeNum)),

            output as (select characterName,
                        Count(Distinct Concat(seasonNum,episodeNum)) as episodeCount,
                        Sum(sceneLength) as screenTime from character_scene_times
                        GROUP BY characterName)

select * from output ORDER BY screenTime DESC;
```

```
   mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
577 rows affected.
Returning data to local variable episode_count_screen_time
```

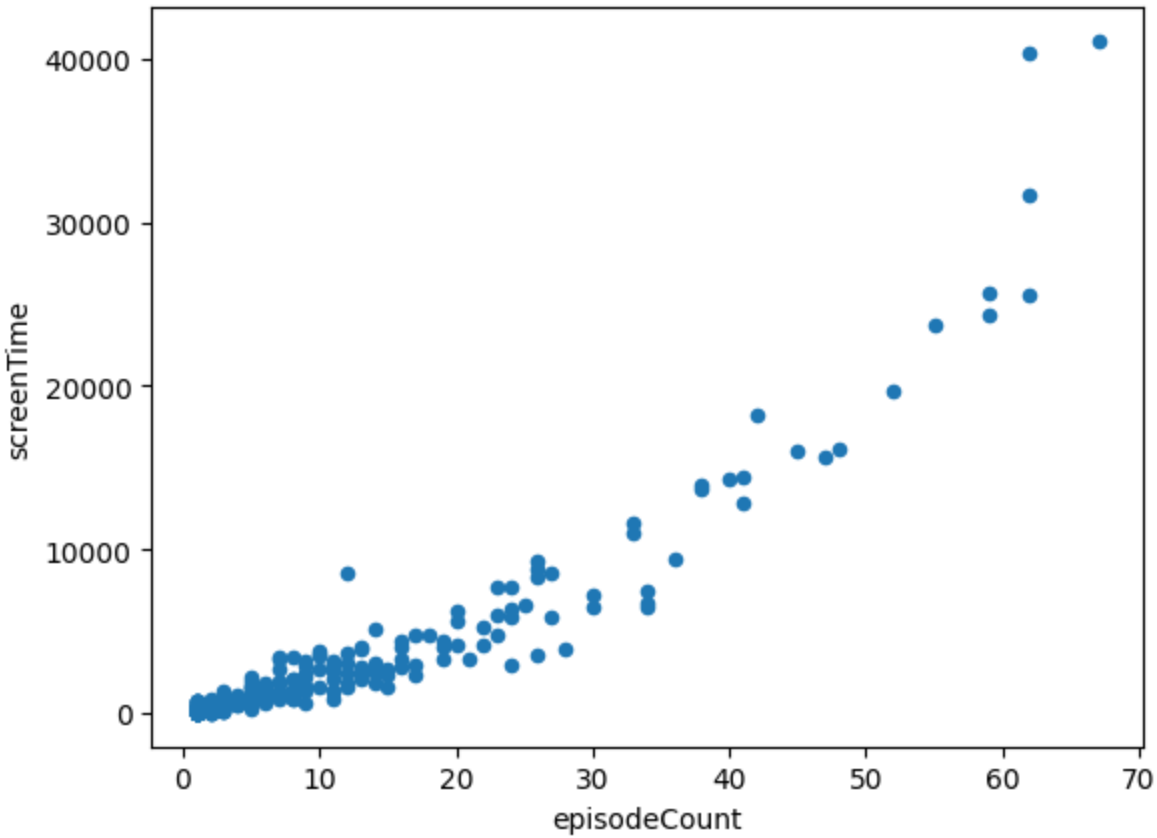
In [40]:
```
# SQL output
# Output is big, so just show first 10 rows

episode_count_screen_time = episode_count_screen_time.DataFrame()
episode_count_screen_time.head(10)
```

Out[40]:

| | characterName | episodeCount | screenTime |
|---|---|---|---|
| **0** | Tyrion Lannister | 67 | 41104 |
| **1** | Jon Snow | 62 | 40365 |
| **2** | Daenerys Targaryen | 62 | 31694 |
| **3** | Sansa Stark | 59 | 25705 |
| **4** | Cersei Lannister | 62 | 25522 |
| **5** | Arya Stark | 59 | 24315 |
| **6** | Jaime Lannister | 55 | 23675 |
| **7** | Jorah Mormont | 52 | 19653 |
| **8** | Davos Seaworth | 42 | 18185 |
| **9** | Samwell Tarly | 48 | 16118 |

In [41]:
```
# TODO: visualization
episode_count_screen_time.plot.scatter(x='episodeCount',y='screenTime');
```

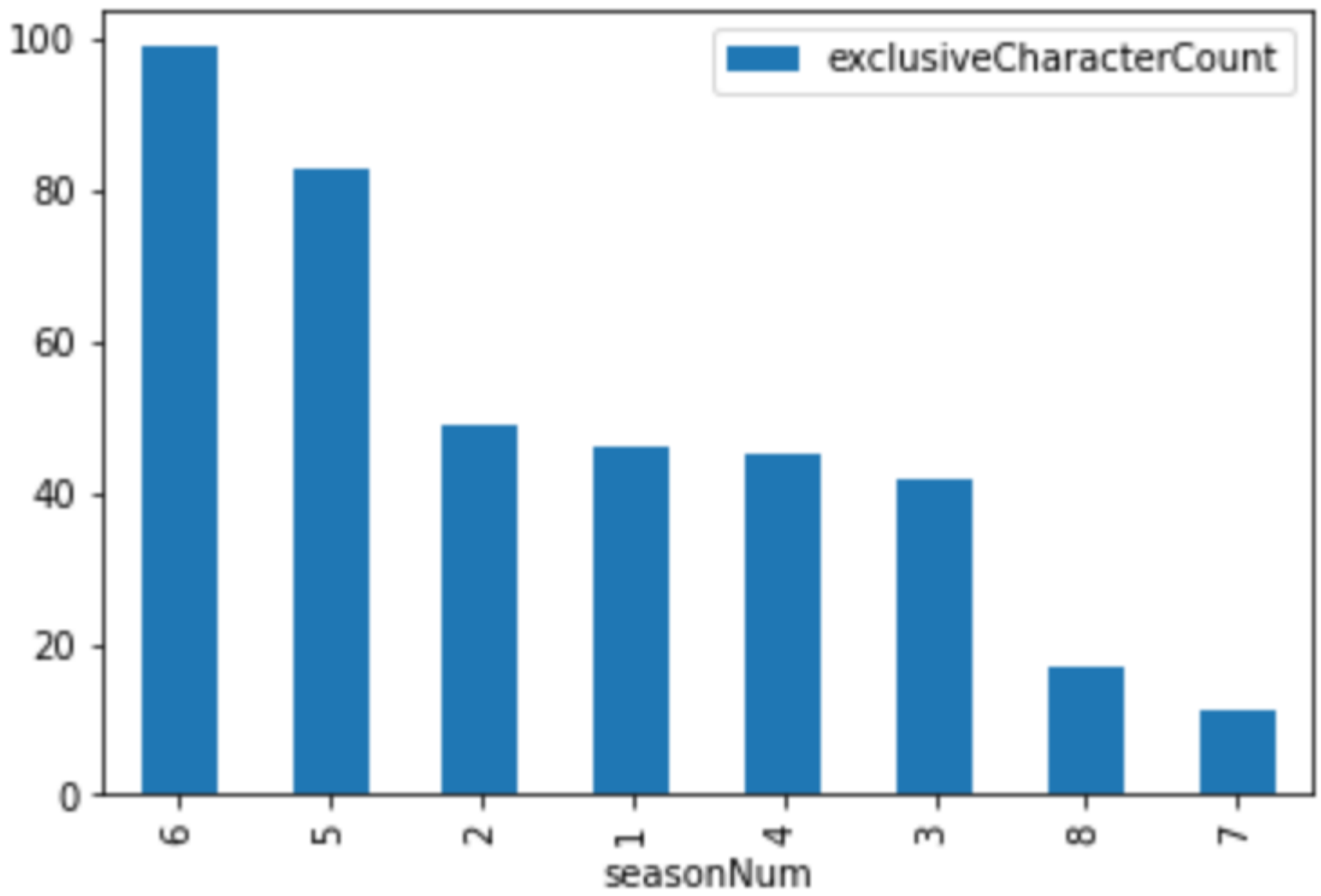



## DV4

- Create a bar chart showing the number of exclusive characters in each season
  - An exclusive character is a character that appeared in only that season, no other season
  - You should order your chart on the number of exclusive characters descending
- You should use the `episodes_characters` table
  - You can assume `characterName` is unique across all characters. That is, a single name is one unique character.
- As an example, your bar chart may look like this:

In [42]:
```
Image("./images/NP5.png")
```

Out[42]:




In [43]:

```
%%sql

season_exclusive_characters <<

WITH exclusive_characters as (select characterName, COUNT(DISTINCT seasonNum) as seasons
                              from episodes_characters
                              GROUP BY characterName
                              HAVING seasons = 1),
     season_counts as (select seasonNum, COUNT(DISTINCT characterName) as exclusiveCharacterCount
                       from episodes_characters
```

```
                                join exclusive_characters using (characterName)
                        GROUP BY seasonNum)
select *
from season_counts
ORDER BY exclusiveCharacterCount DESC;
```

```
  mysql+pymysql://general_user:***@localhost/s24_hw3
  mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
8 rows affected.
Returning data to local variable season_exclusive_characters
```

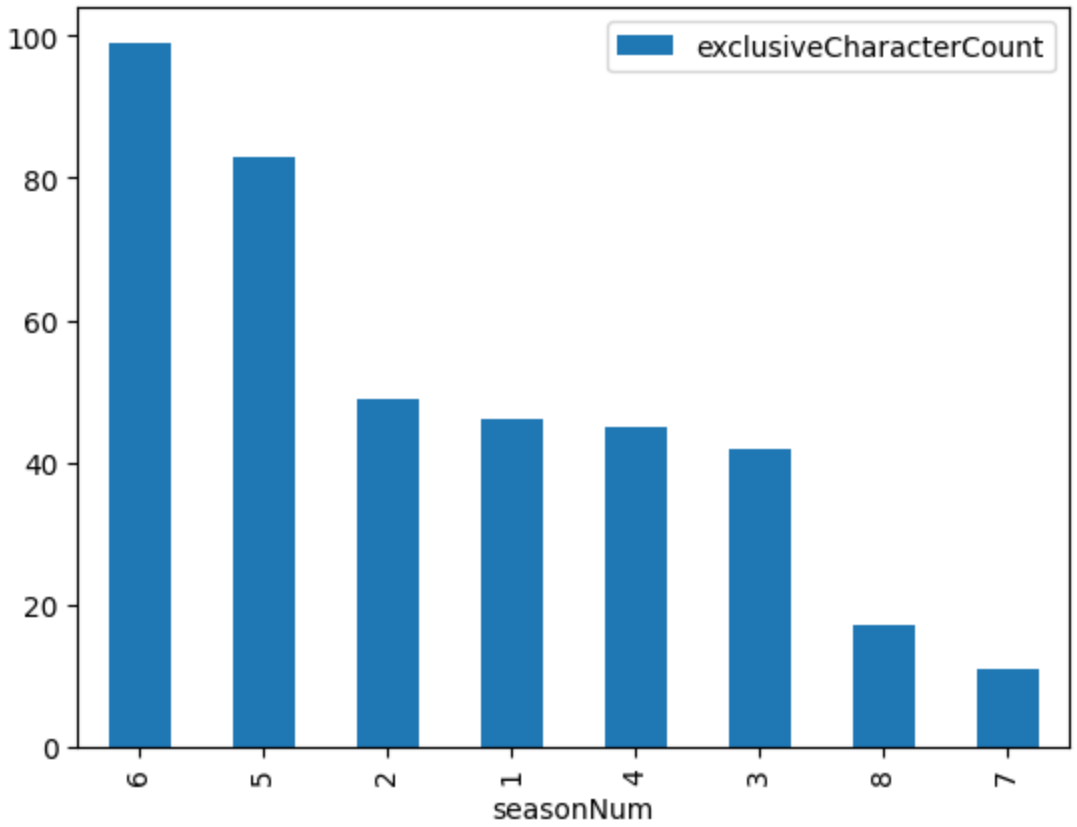
In [44]:
```
# SQL output

season_exclusive_characters = season_exclusive_characters.DataFrame()
season_exclusive_characters
```

Out[44]:

| | seasonNum | exclusiveCharacterCount |
|---|---|---|
| **0** | 6 | 99 |
| **1** | 5 | 83 |
| **2** | 2 | 49 |
| **3** | 1 | 46 |
| **4** | 4 | 45 |
| **5** | 3 | 42 |
| **6** | 8 | 17 |
| **7** | 7 | 11 |

In [45]:
```
# TODO: visualization
season_exclusive_characters.plot.bar(x='seasonNum');
```


exclusiveCharacterCount
100
80
60
40
20
0
6
5
2
1
4
3
8
7
seasonNum